1 General Safety Criteria

To evaluate the suitability of C++ for safety-critical applications [Bin95] and to formulate guidelines for its use in these, the concept of a safety-critical application must first be defined. A safety-critical application [HIS99, Saf00] is one where failure to stay within the bounds of safe operation leads to unacceptable results. Unacceptable results are those which pose a threat to human life, lead to damage of equipment, to excessive pollution, ... To formalize acceptability – to allow quantitative assessment and support decision-making – the impact is typically modelled as a weighted dollar cost. If this cost exceeds a threshold (often imposed by law), behaviour is deemed unacceptable.

Hecht and Brill [HB97] propose a conceptual framework which starts from generic attributes defining the general safety-related quality of software and which leads to language-specific guidelines. The generic attributes are

- **Reliability**: defined as the probability of successful execution under conditions specified in the design. This is closely related to the concept of predictability. Both values obtained from running the software and execution times should be predictable. The boundedness of memory used also has a great impact on reliability. The weakest timing constraint is that of termination of (sub-) programs in finite time. Harder constraints are usually needed in real-time reactive systems which have to react to external events in bounded time.
- **Robustness** or survivability: refers to the capability of software to continue execution after/during abnormal conditions. This attribute is closely related to the handling of exceptions. A typical exception occurs when memory cannot be allocated.
- **Traceability** relates to the ability to identify the code in the context of a disciplined software development process [DOD95]. In particular, the ability to relate requirements and design to code derived from them.
- **Maintainability** is the degree to which the source code reduces the likelihood of fault introduction after delivery (during software maintenance).

In the sequel, all remarks are related to reliability. When stated explicitly, they are also related to other attributes.

It should be noted that a symbolic analysis of source code, attempting to statically prove run-time properties of the code, is orthogonal to language features. It is suggested here that suitability for safety critical operation may be more related to the ability to analyze the program/model than to the language itself.

2 C++ as a Language

Most of the C++ safety issues stem from the safety problems with C [Hat94]. As a general remark, safer C++ constructs (such as new) should be used instead of their unsafe C counterparts (such as malloc).
The complexity of the C++ language [Str97, ES90] has increased dramatically since its early days. Though ANSI standardization has removed most ambiguities, the language has lost simplicity and it is possible to write code which is extremely hard to comprehend. In the context of safety critical software, one should try to maximize simplicity, possibly using only a *subset* of C++'s capabilities. When in doubt whether to use an advanced C++ feature, the ability to guarantee safe behaviour should be the guiding principle. In particular, the static provability of correct behaviour is the primary goal.

Below are some remarks. For a more detailed overview, see [NRC97].

*(Related to traceability)* Comments in source code are a means to

- explain the *meaning* of a program construct which is not apparent from syntax alone. Whenever possible, one should try to use constructs in the language itself to reveal meaning (through judicious use of identifier names and type definitions).
- provide a link between requirements and design specifications on the one hand and implementation on the other hand. The specifications should be as rigorous as possible. Ideally, both code and comments would be *generated* automatically from the specifications.

Though appropriate use of comments in source code will increase reliability, comments have the following drawbacks compared to language constructs proper

- comments are not type-checked nor is their content executed at run-time. Putting an *assertion* or a class method in the source code which checks whether a variable v becomes negative is preferred over a comment /*v should always be >= 0*/.
- as both design and code evolve over time, comments—providing a link to the design—may not be updated, resulting in discrepancies.

The C++ language has a sufficiently rich structure, and the object-oriented programming style encourages “lots of small functions” to allow one to minimize the use of comments.

Dynamic method binding (polymorphism) entails a run-time overhead but is safe (unlike dynamic loading of libraries) as the compiler can statically type-check the code. In hard real-time applications, the time delay caused by Virtual Function Table (VFT) lookup may however prove prohibitive.

Operator overloading is very useful when used consistently and when its meaning is intuitively clear. It becomes dangerous if the intent is not understood directly. In this context, operator precedence is not always intuitive. Hence, parentheses should always be used. A more conservative approach is the use of explicit operator names (function calls) rather than operator overloading.

For a class that defines the operators operator->, operator*, and operator[], the equivalence between p->m, (*p).m, and p[0].m should be ensured.

Literals (#defined constants) should be avoided as they are not type-checked. *consts* or enumerated types should be used instead.

The ANSI C++ provides for namespaces. The use of global variables should be minimized. In the same spirit, class attributes and functions should be made *private* as much as possible.

Dynamic memory allocation should be minimized. Use of C’s *malloc* and *free* should be avoided. Instead, C++'s *new* and *delete* should be employed. *new_handler* must be set explicitly using *set_new_handler* as the default handler will terminate the program when it cannot satisfy a request for memory.

To avoid memory leaks, class constructors

```cpp
foo::foo(a,b): a(...), b(...) {...}
```

must be used rather than

```cpp
foo::foo(a,b): {a=...; b=...; ...}
```

The first will properly call the destructors for a and for b in case the constructor fails. This important issue is discussed in detail in section 14.4.1 of [Str97].
Variables should always be initialized before use. For class constructors, this implies `operator=` should initialize all class attributes.

Multiple inheritance should be avoided as it makes static proving difficult (which member functions are included?). Virtual base classes should be used where possible.

The `class` construct provides far better and safer access control than `struct`. Similarly, a class hierarchy with virtual functions rather than `union` provides type checking of the data stored in the union. `unions` are unsafe as they are untagged, so the type checker cannot assess their correct use.

As Ada [Joh97] was designed with safety critical systems in mind, it is worthwhile to investigate some Ada features [NRC97].

- Ada was designed to allow static provability. Many features (present in C++) were not included as they would jeopardize provability.
- Ada’s type-system is far more conservative than C++’s. Thus, it allows for more detailed checking by the compiler.
- The distinction (in the language itself, not through clever use of include files) between interfaces and implementations allows for safer re-use of different packages.
- Ada’s predefined exceptions and pragmas make exceptions and their semantics part of the language.
- Object-oriented features were added in Ada-95 through (generic) packages. From a language point of view, this is not as elegant as the C++ approach.

As C++ was not designed with safety-critical applications in mind per se, it is much more “liberal” than Ada. It is however not inconceivable to emulate many of the Ada features by combining coding rules and guidelines with a library of “safe” classes. Such safe classes would encode (run-time) tests as well as force (compile-time) type-checks.

## 3 C++ as an Environment

Just as other programming languages, C++ is used as a means to specify desired program behaviour. The actual execution of the program is a result of compilation, linking of program parts, linking with external libraries, and execution in a particular run-time environment.

(Related to traceability) As the C++ language definition does not distinguish between interfaces and implementations, meaningful linking of separately compiled components is difficult (and a source of many errors). In C and C++, judicious use of include files and of the preprocessor are needed.

Different compilers will mangle overloaded function names and class method names differently. Problems will occur when linking binary objects produced by different compilers. One should always use the same compiler for all components. Moreover, all components should be compiled on the same system, with the same libraries. This is particularly relevant for embedded code, where cross-platform development is common. To avoid the problems with name mangling, one may use `extern "C"` to specify C linkage rules. Thus, one needs to wrap overloaded functions and class methods to obtain unique names. Note however how C-style linkage allows for far less type checking than C++-style linkage (in conjunction with well-designed header files).

(Related to maintainability) The state values of software can depend on the particular implementation of compilers and libraries. In particular, in C++ `sizeof(short) <= sizeof(int) <= sizeof(long)` and no particular number of bits is guaranteed. Similarly, one must check that the C++ floating point data type complies with IEEE standards.

(Related to maintainability) External libraries are a safety risk. As they are external (and due to the lack of Ada-style packages) the user’s knowledge of their behaviour is minimal. In particular, libraries may evolve independently of the application software. Above all, statically linked libraries must be used. Dynamic linking saves space, as only what is needed is loaded, only when it is needed. Here, the software writer has hardly any
control over which library will be loaded on the target system.

*(Related to robustness)* Exception handling in C++ is slow and its timing is hard to predict. It is thus best to avoid extensive use of it.

### References


